マッチ棒問題（発表資料）

2013.06.06

１．実測値による性能比較

1.1　測定環境について

今回の測定結果は、すべて下記のデスクトップ PC を使用したものである。

【PC】	【開発環境】
CPU:Intel Core i7 2600 3.4 GHz	コンパイラ：Visual C++ 2008 Express
キャッシュ容量：8 MB	
メモリ容量：4 GB	
グラフィックスドライバ：270.81	
OS :Windows7 Home premium （32ビット版）	

１．２　問題２の（２）：ｒ本のマッチ棒を孤立点がないように取り除いたときの配置の数

(本プログラムは課題とはなっていないが、関連性があるので記載)

【MT231.c】

M　N　R	配置数	探索時間(秒)
4　4　9	4975368	0.63
4　4　10	7082522	1.25
4　4　11	8502064	2.04
5　5　7	78494904	3.10
5　5　8	1382703980	16.61
5　5　9	1382703980	72.72

【les07-ex02-HPI】状態遷移マトリクスを用いたオリジナルプログラム

---- (10,10) ----　　137202 msec

0	1
1	212
2	22 278
3	1 547 004
4	79 847 833
5	3 266 980 696
6	110 355 685 974
7	3 164 929 415 252
8	78 655 530 649 408
9	1 720 472 712 642 520
10	33 529 780 127 200 800
11	587 972 655 594 333 440
12	9 352 846 816 868 387 669
13	135 871 476 036 610 693 224
14	1 812 998 854 989 003 399 836
15	22 329 610 241 912 674 472 796
16	254 929 712 696 202 762 837 897

les07-ex02 は、探索ではなく、状態遷移マトリクスを使った行列演算処理によって配置数を求めている。

1.3　問題３(課題プログラム)

【MT311】

M	N	R	S	配置数	探索時間(秒)
4	4	8	0	0	1.08
4	4	9	0	16	1.95
5	5	8	0	0	183.04
5	5	9	0	0	690.24

【各プログラムの実測結果サマリ】

表１　主なプログラムの測定結果一覧

	3	4	5	6	7	備考
MT311		(1.95)				(0,9)のみ
base model	0.19	0.52	890.42			
LBE0	0.17	0.30	184.94			LBE0
mod10	0.16	0.17	99.19			LBE1 = R + S
mod11	0.16	0.12	12.90	2931.23		LBE2 = 2R + S
mod14	0.16	0.78	9.06	2026.27		LBE3 = LBE1+LBE2
mod50	0.11	0.31	3.59	678.82		mod14+SymCheck、SQC
mod100	0.01	0.05	7.41	172.71		HJ base model
mod102	0.00	0.03	4.65	50.98		mod100 + SQC
mod104	0.00	0.03	4.54	31.26		mod102 + SQC refine
mod110	0.02	0.02	3.95	27.00		mod104 + Mapsソート
mod150	0.02	0.02	3.15	64.48		mod100 + SymCheck
mod154	0.02	0.02	1.90	17.24		mod104+SymCheck
mod160	**0.02**	**0.02**	**1.76**	**15.43**		**mod110+SymCheck**

- base model は、MT311 より 1000 倍以上高速
- 最速の LBE 版(mod50)の改善率　：　248 倍(base model との比較)
- 最速の HJ 版(mod160)の改善率　：　44 倍(最速 LBE 版との比較)

【最速データ詳細】les07-ex03-mod160-

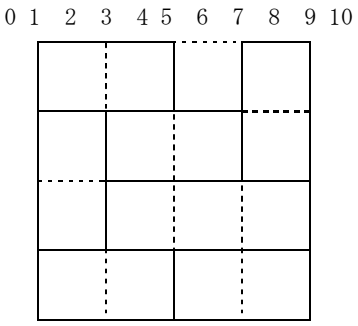
```
    M   N     S   R
---[ 6,  6] / ( 0,19)  :       16   (     1761)          15 msec
---[ 6,  6] / ( 1,19)  :    13632   (   361198)         265 msec
---[ 6,  6] / ( 2,18)  :     2752   (   323816)         203 msec
---[ 6,  6] / ( 3,17)  :       72   (   116834)          78 msec
---[ 6,  6] / ( 4,17)  :   371612   ( 2678292)         780 msec
---[ 6,  6] / ( 5,16)  :    20300   (   569214)         171 msec
---[ 6,  6] / ( 6,16)  :  4239420   ( 6960858)        1653 msec
---[ 6,  6] / ( 7,15)  :   161024   ( 1157496)         281 msec
---[ 6,  6] / ( 8,15)  : 11181548   (10533220)        2667 msec
---[ 6,  6] / ( 9,14)  :   278384   ( 1546066)         343 msec
---[ 6,  6] / (10,14)  : 11129340   (12316203)        3136 msec
---[ 6,  6] / (11,13)  :   165500   ( 1711480)         358 msec
---[ 6,  6] / (12,13)  :  5117504   (12948555)        2730 msec
---[ 6,  6] / (13,12)  :    37360   ( 1764725)         312 msec
---[ 6,  6] / (14,12)  :  1138828   (13127423)        2090 msec
===[ 6,  6]                                  total   15428 msec

---[ 7,  7] / ( 0,26)  :      544   (   163767)        2308 msec
---[ 7,  7] / ( 1,25)  :      248   (   167448)        1170 msec
---[ 7,  7] / ( 2,24)  :       16   (    54288)         390 msec
---[ 7,  7] / ( 3,24)  :    51436   ( 4699243)      127779 msec
---[ 7,  7] / ( 4,23)  :     2232   (   703693)        6458 msec
```

時間がかかるので、中断

２．解法プログラムの概要

2.0　データ表現

１）マッチ棒の配置の表現（Ｍ行Ｎ列、Ｍ≦Ｎ）



１）マッチ棒の上から下への並びを段と呼ぶことにする。各段には 0～2N+2 の番号をつける。
２）マッチ棒の有無を１：０で表現し、各段毎のビットマップで管理する。
３）奇数段を列、偶数段を行と呼ぶ。
４）列のビットマップは、０～ $2^M - 1$個の 2^M 個のパターンが存在する。
５）行のビットマップは、０～ $2^{M+1} - 1$個の 2^{M+1} 個のパターンが存在する。
６）配置全体は、Pos[]に各段のビットマップを入れて管理する。

３）Tips

接続性テスト：setsuzoku(I, J, K)

ある列のビットパターンを J 、その前後の行のビットパターンを I, K とする。

【判定方法】

```
even = (I & K) | (J & (J<<1));        //  1 の数が偶数なら 1 、そうでなければ 0
odd  = (I ^ K) ^ (J ^ (J<<1));        // 1 の数が奇数なら 1 、そうでなければ 0
err  = (odd & ~even) & ColMask;   // odd には 1 が 3 個の場合が含まれるのでそれを除外
err = 0 なら接続性テスト OK
```

正方形の数のカウント：SquareCount(u, row, col)

i 段~ k 段で構成される大きさ u = (k － I) / 2 の正方形のカウント

```
row  = I & K （ビットマップの両方が 1 のものに絞る）
col  = i+1 行から k－1 行までの and 積(ビットマップのすべてが 1 のものに絞る)
```

以下の手順で、u, row, col から正方形の数をカウントする。

```
low  = col & (－col )   col の LSB を求める   low は正方形の下辺
high = low  <<  u       正方形の上辺
col & high == 0 なら、low を底辺とする正方形は成立しない。
c = (high－low)         c は左辺・右辺のビットマップ
row & c == c なら正方形が成立しているのでカウントする
col ^= low              // low を底辺とする正方形を除外
col が 0 になるまで上記を繰り返す。
```

2.1 base model

【基本アルゴリズム】

１）全体の処理構造：第 1 段～第 2N+1 段まで、各段のパターンを再帰的にテストしていく。その時、取り除いたマッチ棒の数(r)、正方形の数(s)を累積していく。もし、r, s のいづれかが上限値 R,S を超えた場合、探索を中止する。(バックトラック)

２）奇数段(列)の処理：

列データのパターンをもとに以下の処理を順次テストする。

・列データをもとに取り除かれたマッチ棒の数を累積する
・追加された列データによって完成する正方形の数を求め（SquareCount）累積する

３）偶数段(行)の処理：

行データのパターンをもとに以下の処理を順次テストする。

・行データをもとに取り除かれたマッチ棒の数を累積する。

・追加された行データによって接続性テストの確認をする。

４） ２N+2 段目の処理：

最終列データを０として、接続性テストの確認をし、ｒ、ｓが上限値と一致していることを確認する。

もし、問題がなければ、解のカウントをする。

３．下限予測値 LBE（Lower Bound Estimater）による探索量の削減

3.1　LBE0

ｎ段までの行の探索が完了した時点を考える。

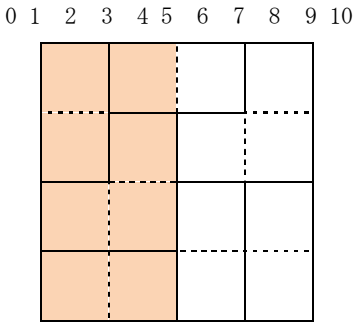
[説明の為の記号]

Ｒ　：取り除くマッチ棒の総数

Ｓ　：正方形の総数

ｒ　：ｎ段までに取り除いたマッチ棒の数

ｓ　：ｎ段までに含まれる正方形の数



↑ここまで探索済（n=4）

（Ｎ－ｎ／２）＊Ｍ は、(n+1)段以降に存在する１ｘ１の正方形の数

ss＝（Ｎ－ｎ／２）＊Ｍ －（Ｓ－ｓ）は、少なくとも削除しなければならない１ｘ１の正方形の数。

一方、(Ｒ－ｒ）本のマッチ棒で削除できる１ｘ１の正方形は最大で（Ｒ－ｒ）＊２ 個であるから、

ss ＞（Ｒ－ｒ）＊２ の条件が成立した時点で探索を中止してよい。

（これ以上続けても、正方形の数は上限値Ｓをオーバーするので）

この極めて単純な LBE0 を base model に追加したのが LBE0 版である。 結果は表１にあるように、**約 4.8 倍の改善**となった。(実際のコードは２行追加しただけであるが)

3.2　LBE1,2,3 （簡易探索による LBE の作成）

【R+S】（RSband）

取り除くマッチ棒の数 R を多くすると、正方形の数 S は少なくなる。逆に、S を大きくとろうとすると、R は小さくせざるをえない。そこで、**σ ＝Ｒ＋Ｓ を評価値**として考える。σ は探索を進めるに従って、単調に増加していくことが予測される。しかし、どのパスを経由してきたかによって、σ の値は様々な値をとるであろう。そこで、ある時点(i 段のパターン I 経由で j 段のパターン J)に至る **σ の最小値を RSband** と定義する。LBE として使用するために、RSband はゴールから逆にたどるパスで σ を計算することにする。

そうすると、ある時点まで探索したとき、

Ｒ　：取り除くマッチ棒の総数

Ｓ　：残すべき正方形の総数

ｒ　：ｎ段までに取り除いたマッチ棒の数

ｓ　：ｎ段までに残された正方形の数

残された評価値：φ＝(Ｒ－ｒ)＋(Ｓ－ｓ)

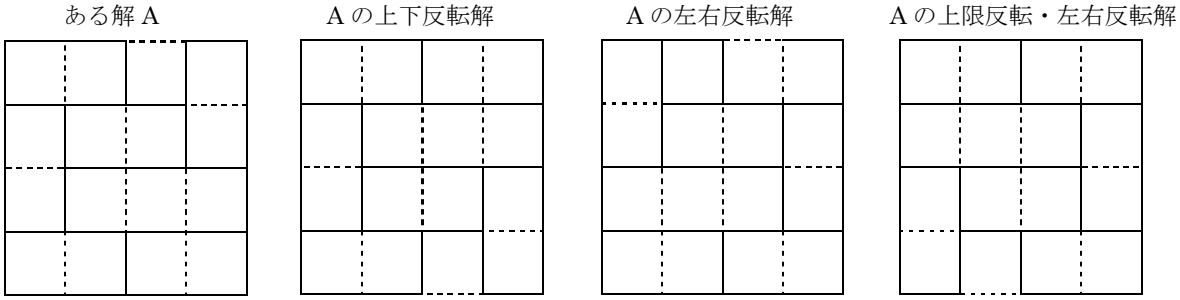
ある時点からゴールまでの評価値　：σ

とすると、　φ　＜　σ であれば、探索を続ける意味はなくなる。

４．対称性を利用した探索量の削減

4.1 解の区分

下図のように、ある解に対称変換（上下反転、左右反転など）を施すことによって、一般には４つ１組の解が得られる。M=N の場合には、更に、９０° 回転操作が加わって、８つ組の解が得られる。(稀なケースとして、これらの操作によって別解が得られない場合もある)



この様な解は、どれか１つ (代表解ととする）を求めれば、他の解は対称変換によって得ることができる。

4.2　代表解の判定条件

深さ半分までの探索を実行した時点で、以下の判定を行う。

条件１：(左右反転解の判定)

左半分のまでの探索で取り除いたマッチ棒の数を R_L

右半分で取り除くであろうマッチ棒の数を R_R （R_R は、R_L 他より求めることができる）

$R_L < R_R$ ：代表解 （得られた解の数を 2 倍として計上）

$R_L = R_R$ ：判定不明 （従来どおり、1 個として計上）

$R_L > R_R$ ：代表解ではないので探索を中止する

条件２：(上下反転解の判定)

中央行(または列)のビットマップを C、C の上下のビットを入れ変えたものを $\tilde{C}$ とする。

$C < \tilde{C}$ ：代表解 （得られた解の数を 2 倍として計上）

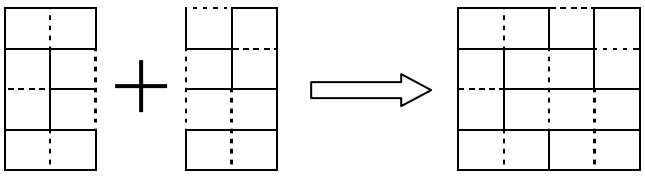
$C = \tilde{C}$ ：判定不明 （従来どおり、1 個として計上）

$C > \tilde{C}$ ：代表解ではないので探索を中止する

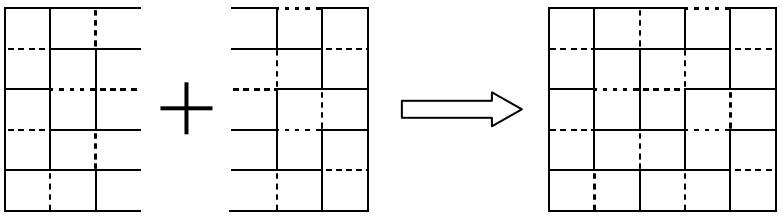
５．部分解合成法：部分解から全体解を生成(Half Join)

M x N の配置を 2 分割した部分解(部品)から全体解を合成する方式である。

(1) N が偶数の場合

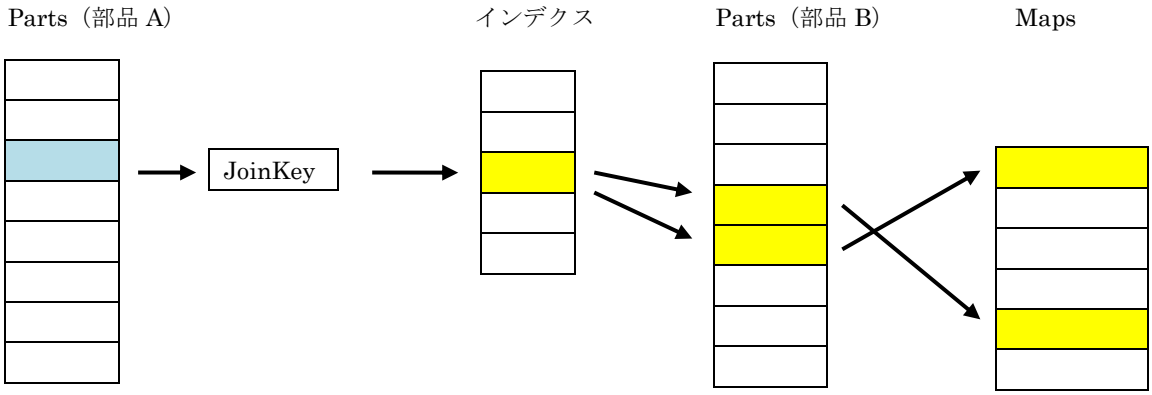


(2) N が奇数の場合



5.3　Join 処理

【テーブルの利用イメージ】



【結合条件と JoinKey】

結合処理に必要な情報を整理するため、結合条件を明確にする。

①中央段のビットパターンが一致すること

②取り除くマッチ棒の数の合計が上限値 R となること

③結合後の正方形の総数が、上限値 S となること

枝刈を行うため、部品内の正方形の数も用意する　→ ③’

（最終的にはすべての Pos[]データが必要）

④N が偶数のとき、中央列とその前後の列に関して、接続性が保証されること

これをもとに、最初のデータ構造を以下の様に決定した。(結果をみながら徐々に見直す方針)

JoinKey　：Hash Join を行うため、①，②、③’の情報をキーとするインデクスを作成する。

①	②	③’

①：map 中央段のビットマップ　　(M+1 ビット)

②：r　　取り除かれたマッチ棒の数　　(6 ビット)

③’：s　　各部品に含まれる正方形の数（4 ビット)